First steps
8.4 Classification ..................................................... 45
8.5 Evaluation ...................................................... 52
8.6 Datasets .......................................................... 54

9 Contributing ....................................................... 57
9.1 Testing ........................................................... 57
9.2 Performance ..................................................... 57

10 Release notes ...................................................... 59
10.1 Version 0.9.0 .................................................. 59
10.2 Version 0.8.1 .................................................. 59
10.3 Version 0.8.0 .................................................. 59
10.4 Version 0.7.2 .................................................. 60
10.5 Version 0.7.1 .................................................. 60
10.6 Version 0.7.0 .................................................. 61
10.7 Version 0.6.0 .................................................. 61
10.8 Version 0.5.0 .................................................. 62
10.9 Version 0.4.0 .................................................. 62
10.10 Version 0.3.0 ................................................ 62
10.11 Version 0.2.0 ................................................ 63
10.12 Version 0.1.0 ................................................ 63

Bibliography ......................................................... 65
All you need to start linking records.
Introduction

The Python Record Linkage Toolkit is a library to link records in or between data sources. The toolkit provides most of the tools needed for record linkage and deduplication. The package contains indexing methods, functions to compare records and classifiers. The package is developed for research and the linking of small or medium sized files.

The project is inspired by the Freely Extensible Biomedical Record Linkage (FEBRL) project, which is a great project. In contrast with FEBRL, the recordlinkage project makes extensive use of data manipulation tools like pandas and numpy. The use of pandas, a flexible and powerful data analysis and manipulation library for Python, makes the record linkage process much easier and faster. The extensive pandas library can be used to integrate your record linkage directly into existing data manipulation projects.

One of the aims of this project is to make an extensible record linkage framework. It is easy to include your own indexing algorithms, comparison/similarity measures and classifiers. The main features of the Python Record Linkage Toolkit are:

- Clean and standardise data with easy to use tools
- Make pairs of records with smart indexing methods such as blocking and sorted neighbourhood indexing
- Compare records with a large number of comparison and similarity measures for different types of variables such as strings, numbers and dates.
- Several classifications algorithms, both supervised and unsupervised algorithms.
- Common record linkage evaluation tools
- Several built-in datasets.

What is record linkage?

The term record linkage is used to indicate the procedure of bringing together information from two or more records that are believed to belong to the same entity. Record linkage is used to link data from multiple data sources or to find
duplicates in a single data source. In computer science, record linkage is also known as data matching or deduplication (in case of search duplicate records within a single file).

In record linkage, the attributes of the entity (stored in a record) are used to link two or more records. Attributes can be unique entity identifiers (SSN, license plate number), but also attributes like (sur)name, date of birth and car model/colour. The record linkage procedure can be represented as a workflow [Christen, 2012]. The steps are: cleaning, indexing, comparing, classifying and evaluation. If needed, the classified record pairs flow back to improve the previous step. The Python Record Linkage Toolkit follows this workflow.

See also:


How to link records?

Import the recordlinkage module with all important tools for record linkage and import the data manipulation framework pandas.

```python
import recordlinkage
import pandas
```

Consider that you try to link two datasets with personal information like name, sex and date of birth. Load these datasets into a pandas DataFrame.

```python
df_a = pandas.DataFrame(YOUR_FIRST_DATASET)
df_b = pandas.DataFrame(YOUR_SECOND_DATASET)
```

Comparing all record can be computationally intensive. Therefore, we make smart set of candidate links with one of the built-in indexing techniques like blocking. Only records pairs agreeing on the surname are included.

```python
block_class = recordlinkage.BlockIndex('surname')
candidate_links = block_class.index(df_a, df_b)
```

Each candidate_link needs to be compared on the comparable attributes. This can be done easily with the Compare class and the available comparison and similarity measures.

```python
compare = recordlinkage.Compare(candidate_links, df_a, df_b)
compare.string('name', 'name', method='jarowinkler', threshold=0.85)
compare.exact('sex', 'gender')
compare.exact('dob', 'date_of_birth')
compare.string('streetname', 'streetname', method='damerau_levenshtein', threshold=0.7)
compare.exact('place', 'placename')
compare.exact('haircolor', 'haircolor', missing_value=9)
```
This record linkage package contains several classification algorithms. Plenty of the algorithms need trainings data (supervised learning) while some others are unsupervised. An example of supervised learning:

```python
true_linkage = pandas.Series(YOUR_GOLDEN_DATA, index=pandas.MultiIndex(YOUR_MULTI_INDEX))
logrg = recordlinkage.LogisticRegressionClassifier()
logrg.learn(compare.vectors[true_linkage.index], true_linkage)
logrg.predict(compare.vectors)
```

and an example of unsupervised learning (the well known ECM-algorithm):

```python
ecm = recordlinkage.BernoulliEMClassifier()
ecm.learn(compare.vectors)
```
Installation

The easiest way of installing the Python Record Linkage Toolkit is using pip. It is as easy as typing:

```
pip install --user recordlinkage
```

You can also clone the project on Github. The license of this record linkage package is GPLv3.

Dependencies

The following packages are required. You probably have most of it already ;)

- numpy
- pandas (>=0.18.0)
- scipy
- sklearn
- jellyfish: Needed for approximate string comparison and string encoding.
- numexpr (optional): Used to speed up numeric comparisons.
Link two datasets

Introduction

This example shows how two datasets with data about persons can be linked. We will try to link the data based on attributes like first name, surname, sex, date of birth, place and address. The data used in this example is part of Febrl and is fictitious.

First, start with importing the recordlinkage module. The submodule recordlinkage.datasets contains several datasets that can be used for testing. For this example, we use the Febrl datasets 4A and 4B. These datasets can be loaded with the function load_febrl4.

```python
In [2]: import recordlinkage as rl
   from recordlinkage.datasets import load_febrl4
   # set logging level to DEBUG, INFO, WARN, ERROR or FATAL. Default WARN
   rl.logging.set_verbosity(rl.logging.WARN)

The datasets are loaded with the following code. The returned datasets are of type pandas.DataFrame. This makes it easy to manipulate the data if desired. For details about data manipulation with pandas, see their comprehensive documentation http://pandas.pydata.org/.

In [3]: dfA, dfB = load_febrl4()
```

```python
dfA
Out[3]:
  rec_id  given_name  surname  street_number  address_1
rec-1070-org  michaela  neumann       8  stanley street
rec-1016-org  courtney  painter      12  pinkerton circuit
rec-4405-org  charles  green        38  salkauskas crescent
rec-1288-org  vanessa  parr        905  macquoid place
rec-3585-org  mikayla  malloney     37  randwick road
...  ...  ...  ...  ...  ...
rec-2153-org  annabel  grierson     97  mclachlan crescent
rec-1604-org  sienna  musolino     22  smeaton circuit
rec-1003-org  bradley  matthews     2  jondol place
```

Make record pairs

It is very intuitive to compare each record in DataFrame \( dfA \) with all records of DataFrame \( dfB \). In fact, we want to make record pairs. Each record pair should contain one record of \( dfA \) and one record of \( dfB \). This process of making record pairs is also called ‘indexing’. With the \texttt{recordlinkage} module, indexing is easy. First, load the \texttt{FullIndex} class. This class makes a full index on a \( .\text{index}(...) \) call. In case of deduplication of a single dataframe, one dataframe is sufficient as argument.

\begin{verbatim}
In [4]: indexer = rl.FullIndex()
    pairs = indexer.index(dfA, dfB)
\end{verbatim}

WARNING:recordlinkage:The number of record pairs is large. Consider a different indexation algorithm

With the method \texttt{index}, all possible (and unique) record pairs are made. The method returns a pandas. MultiIndex. The number of pairs is equal to the number of records in \( dfA \) times the number of records in \( dfB \).

\begin{verbatim}
In [5]: print (len(dfA), len(dfB), len(pairs))
5000 5000 25000000
\end{verbatim}

Many of these record pairs do not belong to the same person. In case of one-to-one matching, the number of matches should be no more than the number of records in the smallest dataframe. In case of full indexing, \( \min(\text{len}(dfA), \text{len}(N_{dfB})) \) is much smaller than \( \text{len}(pairs) \). The \texttt{recordlinkage} module has some more advanced indexing methods to reduce the number of record pairs. Obvious non-matches are left out of the index. Note that if a matching record pair is not included in the index, it can not be matched anymore.
One of the most well known indexing methods is named *blocking*. This method includes only record pairs that are identical on one or more stored attributes of the person (or entity in general). The blocking method can be used in the `recordlinkage` module.

```python
In [6]: indexer = rl.BlockIndex(on='given_name')
   pairs = indexer.index(dfA, dfB)

   print(len(pairs))

77249
```

The argument ‘given_name’ is the blocking variable. This variable has to be the name of a column in `dfA` and `dfB`. It is possible to parse a list of columns names to block on multiple variables. Blocking on multiple variables will reduce the number of record pairs even further.

Another implemented indexing method is *Sorted Neighbourhood Indexing* (`recordlinkage.SortedNeighbourhoodIndex`). This method is very useful when there are many misspellings in the string were used for indexing. In fact, sorted neighbourhood indexing is a generalisation of blocking. See the documentation for details about sorted neighbourhood indexing.

### Compare records

Each record pair is a candidate match. To classify the candidate record pairs into matches and non-matches, compare the records on all attributes both records have in common. The `recordlinkage` module has a class named `Compare`. This class is used to compare the records. The following code shows how to compare attributes.

```python
In [7]: # This cell can take some time to compute.
   compare_cl = rl.Compare(pairs, dfA, dfB)

   compare_cl.exact('given_name', 'given_name', name='given_name')
   compare_cl.string('surname', 'surname', method='jarowinkler', threshold=0.85, name='surname')
   compare_cl.exact('date_of_birth', 'date_of_birth', name='date_of_birth')
   compare_cl.exact('suburb', 'suburb', name='suburb')
   compare_cl.exact('state', 'state', name='state')
   compare_cl.string('address_1', 'address_1', threshold=0.85, name='address_1');
```

All attribute comparisons are stored in a DataFrame with horizontally the features and vertically the record pairs. The comparison can be found in the `vectors` attribute of the `Compare` class. The first 10 comparison vectors are:

```python
In [8]: compare_cl.vectors

Out[8]:
   given_name  surname  date_of_birth  suburb  state  address_1
rec_id  rec_id
rec-1070-org rec-3024-dup-0  1   0.0  0   0
rec-2371-dup-0           1   0.0  0   0
rec-4652-dup-0           1   0.0  0   0
rec-4795-dup-0           1   0.0  0   0
rec-1314-dup-0           1   0.0  0   0
...  ...       ...   ...   ...  ...
rec-4528-org rec-4528-dup-0  1   1.0  1   1
rec-4887-org rec-4887-dup-0  1   1.0  1   0
rec-4350-org rec-4350-dup-0  1   1.0  1   1
rec-4569-org rec-4569-dup-0  1   1.0  1   1
rec-3125-org rec-3125-dup-0  1   1.0  1   0

   state  address_1
rec_id  rec_id
rec-1070-org rec-3024-dup-0  1   0.0
rec-2371-dup-0           0   0.0
```

### 3.3. Compare records
The last step is to decide which records belong to the same person. In this example, we keep it simple:

In [10]: # Sum the comparison results.
   compare_cl.vectors.sum(axis=1).value_counts().sort_index(ascending=False)

Out[10]: 6.0   1566
   5.0   1332
   4.0    343
   3.0    146
   2.0   16427
   1.0    57435
dtype: int64

In [ ]: compare_cl.vectors[compare_cl.vectors.sum(axis=1) > 3]
state address_1
rec_id rec_id
rec-2371-org rec-2371-dup-0 1 1.0
rec-3024-org rec-3024-dup-0 1 0.0
rec-4652-org rec-4652-dup-0 1 1.0
rec-4795-org rec-4795-dup-0 1 1.0
rec-1016-org rec-1016-dup-0 0 1.0
... ... ...
rec-4528-org rec-4528-dup-0 1 1.0
rec-4887-org rec-4887-dup-0 1 1.0
rec-4350-org rec-4350-dup-0 1 1.0
rec-4569-org rec-4569-dup-0 1 0.0
rec-3125-org rec-3125-dup-0 1 1.0

[3241 rows x 6 columns]

Full code

In [ ]: import recordlinkage as rl
from recordlinkage.datasets import load_febrl4
dfA, dfB = load_febrl4()

# Indexation step
indexer = rl.BlockIndex(on='given_name')
pairs = indexer.index(dfA, dfB)

# Comparison step
compare_cl = rl.Compare(pairs, dfA, dfB)

compare_cl.exact('given_name', 'given_name', name='given_name')
compare_cl.string('surname', 'surname', method='jarowinkler', threshold=0.85, name='surname')
compare_cl.exact('date_of_birth', 'date_of_birth', name='date_of_birth')
compare_cl.exact('suburb', 'suburb', name='suburb')
compare_cl.exact('state', 'state', name='state')
compare_cl.string('address_1', 'address_1', threshold=0.85, name='address_1')

# Classification step
matches = compare_cl.vectors[compare_cl.vectors.sum(axis=1) > 3]
print(len(matches))
CHAPTER 4

Data deduplication

Introduction

This example shows how to find records in datasets belonging to the same entity. In our case, we try to deduplicate a dataset with records of persons. We will try to link within the dataset based on attributes like first name, surname, sex, date of birth, place and address. The data used in this example is part of Febrl and is fictitious.

First, start with importing the recordlinkage module. The submodule recordlinkage.datasets contains several datasets that can be used for testing. For this example, we use the Febrl dataset 1. This dataset contains 1000 records of which 500 original and 500 duplicates, with exactly one duplicate per original record. This dataset can be loaded with the function load_febrl1.

In [1]: import recordlinkage
   from recordlinkage.datasets import load_febrl1

The dataset is loaded with the following code. The returned datasets are of type pandas.DataFrame. This makes it easy to manipulate the data if desired. For details about data manipulation with pandas, see their comprehensive documentation http://pandas.pydata.org/.

In [2]: dfA = load_febrl1()


Out[2]:
     given_name surname street_number address_1    address_2 suburb postcode state date_of_birth soc_sec_id
rec_id
rec-223-org       NaN  waller             6   tullaroop street
rec-122-org  lachlan       berry           69  giblin street
rec-373-org  deakin  sondergeld        48  goldfinch circuit
rec-10-dup-0  kooltuo  canterbury       NaN  maltby circuit
rec-227-org    luke    purdon           23   ramsay place

rec_id
rec-223-org  willaroo    st james     4011     wa  19081209    6988048
rec-122-org      killarney   bittern   4814     qld  19990219    7364009
rec-373-org       kooltuo canterbury  2776     vic  19600210   2635962
Make record pairs

It is very intuitive to start with comparing each record in DataFrame dfA with all other records in DataFrame dfA. In fact, we want to make record pairs. Each record pair should contain two different records of DataFrame dfA. This process of making record pairs is also called ‘indexing’. With the recordlinkage module, indexing is easy. First, load the FullIndex class. This class makes a full index on a .index(...) call. In case of deduplication of a single dataframe, one dataframe is sufficient as input argument.

In [3]: indexer = recordlinkage.FullIndex()
   pairs = indexer.index(dfA)

With the method index, all possible (and unique) record pairs are made. The method returns a pandas.MultiIndex. The number of pairs is equal to the number of records in dfA times the number of records in dfB.

In [4]: print (len(dfA), len(pairs))
   # (1000*1000-1000)/2 = 499500
   (1000, 499500)

Many of these record pairs do not belong to the same person. The recordlinkage toolkit has some more advanced indexing methods to reduce the number of record pairs. Obvious non-matches are left out of the index. Note that if a matching record pair is not included in the index, it can not be matched anymore.

One of the most well known indexing methods is named blocking. This method includes only record pairs that are identical on one or more stored attributes of the person (or entity in general). The blocking method can be used in the recordlinkage module.

In [5]: indexer = recordlinkage.BlockIndex(on='given_name')
   pairs = indexer.index(dfA)

   print (len(pairs))
   2082

The argument ‘given_name’ is the blocking variable. This variable has to be the name of a column in dfA and dfB. It is possible to parse a list of columns names to block on multiple variables. Blocking on multiple variables will reduce the number of record pairs even further.

Another implemented indexing method is Sorted Neighbourhood Indexing (Pairs.sortedneighbourhood). This method is very useful when there are many misspellings in the string were used for indexing. In fact, sorted neighbourhood indexing is a generalisation of blocking. See the documentation for details about sorted neighbourhood indexing.

Compare records

Each record pair is a candidate match. To classify the candidate record pairs into matches and non-matches, compare the records on all attributes both records have in common. The recordlinkage module has a class named Compare. This class is used to compare the records. The following code shows how to compare attributes.

In [6]: # This cell can take some time to compute.
   compare_cl = recordlinkage.Compare(pairs, dfA, dfA)

   compare_cl.exact('given_name', 'given_name', name='given_name')
   compare_cl.string('surname', 'surname', method='jarowinkler', threshold=0.85, name='surname')
All attribute comparisons are stored in a DataFrame with horizontally the features and vertically the record pairs. The comparison can be found in `vectors` attribute of the `Compare` class. The first 10 comparison vectors are:

```
In [7]: compare_cl.vectors.head(10)
Out[7]:
given_name surname date_of_birth suburb  
rec_id rec_id
rec-122-org rec-183-dup-0 1 0.0 0 0
rec-248-org 1 0.0 0 0
rec-469-org 1 0.0 0 0
rec-74-org 1 0.0 0 0
rec-183-org 1 0.0 0 0
rec-360-dup-0 1 0.0 0 0
rec-248-dup-0 1 0.0 0 0
rec-469-dup-0 1 0.0 0 0
rec-183-dup-0 rec-248-org 1 0.0 0 0
rec-469-org 1 0.0 0 0

state address_1  
rec_id rec_id
rec-122-org rec-183-dup-0 0 0.0
rec-248-org 1 0.0
rec-469-org 0 0.0
rec-74-org 0 0.0
rec-183-org 0 0.0
rec-360-dup-0 0 0.0
rec-248-dup-0 1 0.0
rec-469-dup-0 0 0.0
rec-183-dup-0 rec-248-org 0 0.0
rec-469-org 1 0.0
```

The last step is to decide which records belong to the same person. In this example, we keep it simple:

```
In [9]: # Sum the comparison results.
   compare_cl.vectors.sum(axis=1).value_counts().sort_index(ascending=False)
```

### 4.3. Compare records
Out[9]:
6.0  142
5.0  145
4.0  30
3.0   9
2.0  376
1.0 1380
dtype: int64

In [10]: matches = compare_cl.vectors[compare_cl.vectors.sum(axis=1) > 3]

print(len(matches))
matches.head(10)

317

Out[10]:
given_name  surname  date_of_birth  suburb  state  
rec_id  rec_id
rec-183-dup-0  rec-183-org  1  1.0  1  1  1
rec-122-dup-0  rec-122-org  1  1.0  1  1  1
rec-248-dup-0  rec-248-org  1  1.0  1  1  1
rec-373-dup-0  rec-373-org  1  1.0  1  1  1
rec-10-dup-0   rec-10-org   1  1.0  0  1  1
rec-342-dup-0  rec-342-org  1  1.0  0  1  1
rec-330-dup-0  rec-330-org  1  0.0  1  1  1
rec-397-dup-0  rec-397-org  1  1.0  1  1  1
rec-472-dup-0  rec-472-org  1  1.0  1  1  1
rec-190-dup-0  rec-190-org  1  1.0  0  1  1

address_1
rec_id  rec_id
rec-183-dup-0  rec-183-org  1.0
rec-122-dup-0  rec-122-org  1.0
rec-248-dup-0  rec-248-org  1.0
rec-373-dup-0  rec-373-org  1.0
rec-10-dup-0   rec-10-org   1.0
rec-342-dup-0  rec-342-org  1.0
rec-330-dup-0  rec-330-org  0.0
rec-397-dup-0  rec-397-org  0.0
rec-472-dup-0  rec-472-org  0.0
rec-190-dup-0  rec-190-org  1.0

**Full code**

In [11]:
import recordlinkage
from recordlinkage.datasets import load_febrl1

dfA = load_febrl1()

# Indexation step
pcl = recordlinkage.BlockIndex(on='given_name')
pairs = pcl.index(dfA)

# Comparison step
compare_cl = recordlinkage.Compare(pairs, dfA, dfA)

compare_cl.exact('given_name', 'given_name', name='given_name')
compare_cl.string('surname', 'surname', method='jarowinkler', threshold=0.85, name='surname')
compare_cl.exact('date_of_birth', 'date_of_birth', name='date_of_birth')
compare_cl.exact('suburb', 'suburb', name='suburb')
compare_cl.exact('state', 'state', name='state')
compare_cl.string('address_1', 'address_1', threshold=0.85, name='address_1');

# Classification step
matches = compare_cl.vectors[compare_cl.vectors.sum(axis=1) > 3]
print(len(matches))
Record pair classification

In the context of record linkage, classification refers to the process of dividing record pairs into matches and non-matches (distinct pairs). There are dozens of classification algorithms for record linkage. Roughly speaking, classification algorithms fall into two groups:

- **supervised learning algorithms** - These algorithms make use of trainings data. If you do have trainings data, then you can use supervised learning algorithms. Most supervised learning algorithms offer good accuracy and reliability. Examples of supervised learning algorithms in the *Python Record Linkage Toolkit* are *Logistic Regression*, *Naive Bayes* and *Support Vector Machines*.

- **unsupervised learning algorithms** - These algorithms do not need training data. The *Python Record Linkage Toolkit* supports *K-means clustering* and an *Expectation/Conditional Maximisation* classifier.

**First things first**

The examples below make use of the *Krebs register* (German for cancer registry) dataset. The Krebs register dataset contains comparison vectors of a large set of record pairs. For each record pair, it is known if the records represent the same person (match) or not (non-match). This was done with a massive clerical review. First, import the recordlinkage module and load the Krebs register data. The dataset contains 5749132 compared record pairs and has the following variables: first name, last name, sex, birthday, birth month, birth year and zip code. The Krebs register contains $\text{len(krebs\_match)} = 20931$ matching record pairs.

**In [2]:**

```python
import recordlinkage as rl
from recordlinkage.datasets import load_krebsregister
krebs_data, krebs_match = load_krebsregister(missing_values=0)
krebs_data
```

**Out[2]:**

```
  cmp\_firstname1  cmp\_firstname2  cmp\_lastname1  cmp\_lastname2  id1  id2
------------------  ------------------  ------------------  ------------------  ----  ----
   22161  38467       1.00000         0.0         0.14286         0.0
   38713  75352       0.00000         0.0         0.57143         0.0
   13699  32825       0.16667         0.0         0.00000         0.0
   22709  37682       0.28571         0.0         1.00000         0.0
   2342  69060       0.25000         0.0         0.12500         0.0
     ...     ...            ...            ...            ...     ...
  52124  53629       1.00000         0.0         0.28571         0.0
```
Most classifiers cannot handle comparison vectors with missing values. To prevent issues with the classification algorithms, we convert the missing values into disagreeing comparisons (using argument missing_values=0). This approach for handling missing values is widely used in record linkage applications.

In [3]: krebs_data.describe().T

Out[3]:

<table>
<thead>
<tr>
<th>count</th>
<th>mean</th>
<th>std</th>
<th>min</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp_firstname1</td>
<td>5749132.0</td>
<td>0.71278</td>
<td>0.38884</td>
<td>0.0</td>
<td>0.28571</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>cmp_firstname2</td>
<td>5749132.0</td>
<td>0.01623</td>
<td>0.12520</td>
<td>0.0</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>cmp_lastname1</td>
<td>5749132.0</td>
<td>0.31563</td>
<td>0.33423</td>
<td>0.0</td>
<td>0.10000</td>
<td>0.18182</td>
<td>0.42857</td>
</tr>
<tr>
<td>cmp_lastname2</td>
<td>5749132.0</td>
<td>0.00014</td>
<td>0.01008</td>
<td>0.0</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>cmp_sex</td>
<td>5749132.0</td>
<td>0.95500</td>
<td>0.20730</td>
<td>0.0</td>
<td>1.00000</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>cmp_birthday</td>
<td>5749132.0</td>
<td>0.22443</td>
<td>0.41721</td>
<td>0.0</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>cmp_birthmonth</td>
<td>5749132.0</td>
<td>0.48879</td>
<td>0.49987</td>
<td>0.0</td>
<td>1.00000</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>cmp_birthyear</td>
<td>5749132.0</td>
<td>0.22272</td>
<td>0.41607</td>
<td>0.0</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>cmp_zipcode</td>
<td>5749132.0</td>
<td>0.00552</td>
<td>0.07407</td>
<td>0.0</td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

Supervised learning

As described before, supervised learning algorithms do need training data. Training data is data for which the true match status is known for each comparison vector. In the example in this section, we consider that the true match status of the first 5000 record pairs of the Krebs register data is known.

In [4]: golden_pairs = krebs_data[0:5000]
golden_matches_index = golden_pairs.index & krebs_match # 2093 matching pairs
Logistic regression

The `recordlinkage.LogisticRegressionClassifier` classifier is an application of the logistic regression model. This supervised learning method is one of the oldest classification algorithms used in record linkage. In situations with enough training data, the algorithm gives relatively good results.

```
In [5]: # Initialize the classifier
    logreg = rl.LogisticRegressionClassifier()

    # Train the classifier
    logreg.learn(golden_pairs, golden_matches_index)
    print ("Intercept: ", logreg.intercept)
    print ("Coefficients: ", logreg.coefficients)

    Intercept: -6.2980435710064135
    Coefficients: [ 4.90452843e-01 1.21640484e-01 2.15040485e+00 -2.84818101e-03
                   -1.79712465e+00 9.61085558e-01 6.72610441e-02 1.03408608e+00
                   4.30556110e+00]

In [6]: # Predict the match status for all record pairs
    result_logreg = logreg.predict(krebs_data)

    len(result_logreg)

    Out[6]: 20150

In [7]: conf_logreg = rl.confusion_matrix(krebs_match, result_logreg, len(krebs_data))

    conf_logreg

    Out[7]: array([[ 19884,  1047],
                   [  266, 5727935]])

In [8]: # The F-score for this prediction is
    rl.fscore(conf_logreg)

    Out[8]: 0.96804
```

The predicted number of matches is not much more than the 20931 true matches. The result was achieved with a small training dataset of 5000 record pairs.

In (older) literature, record linkage procedures are often divided in deterministic record linkage and probabilistic record linkage. The Logistic Regression Classifier belongs to deterministic record linkage methods. Each feature/variable has a certain importance (named weight). The weight is multiplied with the comparison/similarity vector. If the total sum exceeds a certain threshold, it is considered to be a match.

```
In [9]: intercept = -9
    coefficients = [2.0, 1.0, 3.0, 1.0, 1.0, 1.0, 1.0, 2.0, 3.0]

    logreg = rl.LogisticRegressionClassifier(coefficients, intercept)

    # predict without calling LogisticRegressionClassifier.learn
    matches = logreg.predict(krebs_data)

    print (len(matches))

    21303

In [10]: conf_logreg = rl.confusion_matrix(krebs_match, matches, len(krebs_data))

    # The F-score for this classification is
    rl.fscore(conf_logreg)

    Out[10]: 0.98769
```
For the given coefficients, the F-score is better than the situation without trainings data. Surprising? No (use more trainings data and the result will improve)

**Naive Bayes**

In contrast to the logistic regression classifier, the Naive Bayes classifier is a probabilistic classifier. The probabilistic record linkage framework by Fellegi and Sunter (1969) is the most well-known probabilistic classification method for record linkage. Later, it was proved that the Fellegi and Sunter method is mathematically equivalent to the Naive Bayes method in case of assuming independence between comparison variables.

```python
In [11]: # Train the classifier
    nb = rl.NaiveBayesClassifier()
    nb.learn(golden_pairs, golden_matches_index)

    # Predict the match status for all record pairs
    result_nb = nb.predict(krebs_data)
    len(result_nb)

Out[11]: 20345

In [12]: conf_nb = rl.confusion_matrix(krebs_match, result_nb, len(krebs_data))
    conf_nb

Out[12]: array([[ 20023,  908],
                [  322, 5727879]])

In [13]: # The F-score for this classification is
    rl.fscore(conf_nb)

Out[13]: 0.97020
```

**Support Vector Machines**

Support Vector Machines (SVM) have become increasingly popular in record linkage. The algorithm performs well there is only a small amount of training data available. The implementation of SVM in the Python Record Linkage Toolkit is a linear SVM algorithm.

```python
In [14]: # Train the classifier
    svm = rl.SVMClassifier()
    svm.learn(golden_pairs, golden_matches_index)

    # Predict the match status for all record pairs
    result_svm = svm.predict(krebs_data)
    len(result_svm)

Out[14]: 20839

In [15]: conf_svm = rl.confusion_matrix(krebs_match, result_svm, len(krebs_data))
    conf_svm

Out[15]: array([[ 20825,  106],
                [   14, 5728187]])

In [16]: # The F-score for this classification is
    rl.fscore(conf_svm)

Out[16]: 0.99713
```
Unsupervised learning

In situations without training data, unsupervised learning can be a solution for record linkage problems. In this section, we discuss two unsupervised learning methods. One algorithm is K-means clustering, and the other algorithm is an implementation of the Expectation-Maximisation algorithm. Most of the time, unsupervised learning algorithms take more computational time because of the iterative structure in these algorithms.

K-means clustering

The K-means clustering algorithm is well-known and widely used in big data analysis. The K-means classifier in the Python Record Linkage Toolkit package is configured in such a way that it can be used for linking records. For more info about the K-means clustering see Wikipedia.

```python
In [17]: kmeans = rl.KMeansClassifier()
   : result_kmeans = kmeans.learn(krebs_data)
   :
   #: The predicted number of matches
   : len(result_kmeans)
Out[17]: 371525
```

The classifier is now trained and the comparison vectors are classified.

```python
In [18]: cm_kmeans = rl.confusion_matrix(krebs_match, result_kmeans, len(krebs_data))
   :
   : rl.fscore(cm_kmeans)
Out[18]: 0.10598
```

Expectation/Conditional Maximization Algorithm

The ECM-algorithm is an Expectation-Maximisation algorithm with some additional constraints. This algorithm is closely related to the Naive Bayes algorithm. The ECM algorithm is also closely related to estimating the parameters in the Fellegi and Sunter (1969) framework. The algorithms assume that the attributes are independent of each other. The Naive Bayes algorithm uses the same principles.

```python
In [19]: # Train the classifier
   : ecm = rl.ECMClassifier()
   : result_ecm = ecm.learn((krebs_data > 0.8).astype(int))
   :
   : len(result_ecm)
Out[19]: 19817
```

```python
In [20]: conf_ecm = rl.confusion_matrix(krebs_match, result_ecm, len(krebs_data))
   :
   : conf_ecm
Out[20]: array([[ 19813,  1118],
   : [     4, 5728197]])
```

```python
In [21]: # The F-score for this classification is
   : rl.fscore(conf_ecm)
Out[21]: 0.97246
```
CHAPTER 6

Custom algorithms

The Python Record Linkage Toolkit contains several built-in algorithms for making record pairs and comparing record pairs. Sometimes, these built-in algorithms do not fit your needs. With the Python Record Linkage Toolkit, it is easy to use other algorithms. This section describes how to implement custom algorithms for the making and comparing record pairs. When you think your algorithm might help others, consider sharing it!

For run the examples below, import pandas, recordlinkage and the two datasets belonging to sample dataset FEBRL4.

```python
In [2]: import pandas

import recordlinkage as rl
from recordlinkage.datasets import load_febrl4

df_a, df_b = load_febrl4()
```

Custom index algorithms

The Python Record Linkage Toolkit contains multiple algorithms to pair records (index algorithms) such as full indexing, blocking and sorted neighbourhood indexing. This section explains how to make and implement a custom algorithm to make record pairs.

To create a custom algorithm, you have to make a subclass of the `recordlinkage.base.BaseIndexator`. In your subclass, you overwrite the `_link_index` method. This method accepts two pandas DataFrames as arguments. Based on these DataFrames, your method must create pairs and return them as a pandas.MultiIndex in which the MultiIndex names are the index names of DataFrame A and DataFrame B respectively.

The algorithm for linking data frames can be used for finding duplicates as well. In this situation, DataFrame B is a copy of DataFrame A. The `Pairs` class removes pairs like (record_i, record_i) and one of the following (record_i, record_j) (record_j, record_i) under the hood. As result of this, only unique combinations are returned. If you do have a specific algorithm for finding duplicates, then you can overwrite the `_dedup_index` method. This method accepts only one argument (DataFrame A) and the internal base class does not look for combinations like explained above.

Let's make an algorithm that pairs records of which the given name in both records starts with the letter ‘W’.
In [3]: from recordlinkage.base import BaseIndexator

    class FirstLetterWIndex(BaseIndexator):
        """Custom class for indexing"""
        def _link_index(self, df_a, df_b):
            """Make pairs of all records where the given name start with the letter 'W'."""
            # Select records with names starting with a w.
            name_a_startswith_w = df_a[df_a['given_name'].str.startswith('w') == True]
            name_b_startswith_w = df_b[df_b['given_name'].str.startswith('w') == True]

            # Make a product of the two numpy arrays
            return pandas.MultiIndex.from_product([name_a_startswith_w.index.values, name_b_startswith_w.index.values],
                                                   names=[df_a.index.name, df_b.index.name])

In [4]: indexer = FirstLetterWIndex()
    candidate_pairs = indexer.index(df_a, df_b)

    print ('Returns a', type(candidate_pairs).__name__)  
    print ('Number of candidate record pairs starting with the letter w:', len(candidate_pairs))

Returns a MultiIndex  
Number of candidate record pairs starting with the letter w: 6072

The custom index class below does not restrict the first letter to 'w', but the first letter is an argument (named letter). This letter can is initialized during the setup of the class.

In [5]: class FirstLetterIndex(BaseIndexator):
        """Custom class for indexing"""
        def __init__(self, letter):
            super(FirstLetterIndex, self).__init__()
            # the letter to save
            self.letter = letter

        def _link_index(self, df_a, df_b):
            """Make record pairs that agree on the first letter of the given name.""
            # Select records with names starting with a 'letter'.
            a_startswith_w = df_a[df_a['given_name'].str.startswith(self.letter) == True]
            b_startswith_w = df_b[df_b['given_name'].str.startswith(self.letter) == True]

            # Make a product of the two numpy arrays
            return pandas.MultiIndex.from_product([a_startswith_w.index.values, b_startswith_w.index.values],
                                                   names=[df_a.index.name, df_b.index.name])

In [6]: indexer = FirstLetterIndex('w')
    candidate_pairs = indexer.index(df_a, df_b)

    print ('Number of record pairs (letter w):', len(candidate_pairs))

Number of record pairs (letter w): 6072

In [7]: for letter in 'wxa':

            indexer = FirstLetterIndex(letter)
candidate_pairs = indexer.index(df_a, df_b)

print ('Number of record pairs (letter %s):' % letter, len(candidate_pairs))

Number of record pairs (letter w): 6072
Number of record pairs (letter x): 132
Number of record pairs (letter a): 172431

Custom compare algorithms

The Python Record Linkage Toolkit holds algorithms to compare strings, numeric values and dates. These functions may not be sufficient for your record linkage. The internal framework of the toolkit makes it easy to implement custom algorithms to compare records.

A custom algorithm is a function that accepts two arguments, one pandas.Series with information on the variable in the first file and one pandas.Series with information on the variable in the second file. The function returns a pandas.Series, numpy.array or list with the similarity/comparison values.

In [8]:
def compare_zipcodes(s1, s2):
    """
    If the zipcodes in both records are identical, the similarity is 0. If the first two values agree and the last two don't, then the similarity is 0.5. Otherwise, the similarity is 0.
    """

    # check if the zipcode are identical (return 1 or 0)
    sim = (s1 == s2).astype(float)

    # check the first 2 numbers of the distinct comparisons
    sim[(sim == 0) & (s1.str[0:2] == s2.str[0:2])] = 0.5

    return sim

In [9]:

# Make an index of record pairs
pcl = rl.BlockIndex('given_name')
candidate_pairs = pcl.index(df_a, df_b)

crl = rl.Compare(candidate_pairs, df_a, df_b)

# default algorithm
crl.string('surname', 'surname', method='jarowinkler', name='sim_surname')

# custom zipcodes algorithm
# crl.compare(callable_algorithm, 'label_A', 'label_B', name='sim_name')
crl.compare(compare_zipcodes, 'postcode', 'postcode', name='sim_postcode')

crl.vectors['sim_postcode'].value_counts()

Out[9]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>71229</td>
</tr>
<tr>
<td>0.5</td>
<td>3166</td>
</tr>
<tr>
<td>1.0</td>
<td>2854</td>
</tr>
</tbody>
</table>

Name: sim_postcode, dtype: int64

As you can see, one can pass the labels of the columns as arguments. The first argument is a column label, or a list of column labels, found in the first DataFrame (postcode in this example). The second argument is a column label, or a list of column labels, found in the second DataFrame (also postcode in this example). The recordlinkage. Compare class selects the columns with the given labels before passing them to the custom algorithm/function. The

6.2. Custom compare algorithms
compare method in the recordlinkage.Compare class passes additional (keyword) arguments to the custom function.

**Warning:** Do not change the order of the pairs in the MultiIndex.

**Multicolumn comparisons**

The Python Record Linkage Toolkit supports the comparison of more than two columns. This is especially useful in situations with multi-dimensional data (for example geographical coordinates) and situations where fields can be swapped.

The FEBRL4 dataset has two columns filled with address information (address_1 and address_2). In a naive approach, one compares address_1 of file A with address_1 of file B and address_2 of file A with address_2 of file B. If the values for address_1 and address_2 are swapped during the record generating process, the naive approach considers the addresses to be distinct. In a more advanced approach, address_1 of file A is compared with address_1 and address_2 of file B. Variable address_2 of file A is compared with address_1 and address_2 of file B. This is done with the single function given below.

```python
In [10]: def compare_addresses(s1_1, s1_2, s2_1, s2_2):
    
    """
    Compare addresses. Compare address_1 of file A with address_1 and address_2 of file B. The same for address_2 of dataset 1.
    """

    return (s1_1 == s2_1) | (s1_2 == s2_2) | (s1_1 == s2_2) | (s1_2 == s2_1).astype(float)
```

```python
In [11]: crl = rl.Compare(candidate_pairs, df_a, df_b)
    
    # naive
    crl.exact('address_1', 'address_1', name='sim_address_1')
    crl.exact('address_2', 'address_2', name='sim_address_2')

    # better
    crl.compare(compare_addresses, ['address_1', 'address_2'], ['address_1', 'address_2'], name='sim_address')

Out[11]:
    sim_address_1  0.02488
    sim_address_2  0.02025
    sim_address    0.03566
    dtype: float64
```
Performance plays an important role in record linkage. Record linkage problems scale quadratically with the size of the file(s). The number of record pairs can be enormous and so are the number of comparisons. Over the years, smart techniques are developed to reduce the number of record pairs. The Python Record Linkage Toolkit offers several effective techniques for making record pairs. Nevertheless, the Python Record Linkage Toolkit is NOT developed with speed in mind. Understandability, usability and extensibility are the main focus points in the development. This makes the toolkit very useful for linking small or medium sized files.

Okeee... There is not only bad news. The performance of many algorithms is good. Nevertheless, there are plenty of options to increase the performance of your record linkage implementation. Some methods are discussed in this topic. Do you know more tricks? Let us know!

## Indexation

### Block on multiple columns

Blocking is an effective way to increase the performance of your record linkage. If the performance of your implementation is still poor, decrease the number of pairs by blocking on multiple variables. This implies that the record pair agrees on two or more variables. In the following example, the record pairs agree on the given name and surname.

```python
p = recordlinkage.BlockIndex(left_on=['first_name', 'surname'],
                             right_on=['name', 'surname'])
pairs = p.index(dfA, dfB)
```

You might exclude more links than desired. This can be solved by repeating the process with a different combination of blocking variables. In the end, merge the links of the two, so called, index passes.

```python
p_surname = recordlinkage.BlockIndex(left_on=['first_name', 'surname'],
                                      right_on=['name', 'surname'])
p_age = recordlinkage.BlockIndex(left_on=['first_name', 'age'],
                                 right_on=['name', 'age'])
```
pairs_surname = p_surname.index(dfA, dfB)
pairs_age = p_age.index(dfA, dfB)

# make a union of the candidate links of both classes
pairs_surname.union(pairs_age)

Note: Sorted Neighbourhood indexing supports, besides the sorted neighbourhood, additional blocking on variables.

Make record pairs

The structure of the Python Record Linkage Toolkit has a drawback for the performance. In the indexation step (the step in which record pairs are selected), only the index of both records is stored. The entire records are not stored. This results in less memory usage. The drawback is that the records need to be queried from the data.

Comparing

Compare only discriminating variables

Not all variables may be worth comparing in a record linkage. Some variables do not discriminate the links of the non-links or do have only minor effects. These variables can be excluded. Only discriminating and informative should be included.

Prevent string comparisons

String similarity measures and phonetic encodings are computationally expensive. Phonetic encoding takes place on the original data, while string simililaty measures are applied on the record pairs. After phonetic encoding of the string variables, exact comparing can be used instead of computing the string similarity of all record pairs. If the number of candidate pairs is much larger than the number of records in both datasets together, then consider using phonetic encoding of string variables instead of string comparison.

String comparing

Comparing strings is computationally expensive. The Python Record Linkage Toolkit uses the package jellyfish for string comparisons. The package has two implementations, a C and a Python implementation. Ensure yourself of having the C-version installed (import jellyfish.cjellyfish should not raise an exception).

There can be a large difference in the performance of different string comparison algorithms. The Jaro and Jaro-Winkler methods are faster than the Levenshtein distance and much faster than the Damerau-Levenshtein distance.

Indexing with large files

Sometimes, the input files are very large. In that case, it can be hard to make an index without running out of memory in the indexing step or in the comparing step. recordlinkage has a method to deal with large files. It is fast, although is not primary developed to be fast. SQL databases may outperform this method. It is especially developed for the useability. The idea was to split the input files into small blocks. For each block the record pairs are computed. Then iterate over the blocks. Consider full indexing:
```python
import recordlinkage
import numpy

cl = recordlinkage.FullIndex()

for dfB_subset in numpy.split(dfB):
    # a subset of record pairs
    pairs_subset = cl.index(dfA, dfB_subset)

    # Your analysis on pairs_subset here
```
This page contains the API reference of the Python Record Linkage Toolkit.

## Standardising

Cleaning and standardising your data may increase your record linkage accuracy. The Python Record Linkage Toolkit contains several tools for data cleaning and standardising. Some of the tools included are: phonetic encoding algorithms and string cleaning tools. The tools are included in the submodule `standardise`. (Import example: `from recordlinkage.standardise import clean, phonenumbers`)

```python
clean(s, lowercase=True, replace_by_none='[^\^\^A-Za-z0-9]+', replace_by_whitespace='[\-\_]', strip_accents=None, remove_brackets=True, encoding='utf-8', decode_error='strict')
```

Clean strings in the Series by removing unwanted tokens, whitespace and brackets.

**Parameters**

- `s` (*pandas.Series*) – A Series to clean.
- `lower` (*bool, optional*) – Convert strings in the Series to lowercase. Default True.
- `replace_by_none` (*str, optional*) – The matches of this regular expression are replaced by “”.  
- `replace_by_whitespace` (*str, optional*) – The matches of this regular expression are replaced by a whitespace.
- `remove_brackets` (*bool, optional*) – Remove all content between brackets and the brackets themselves. Default True.
- `strip_accents` (*{'ascii', 'unicode', None}, optional*) – Remove accents during the preprocessing step. ‘ascii’ is a fast method that only works on characters that have an direct ASCII mapping. ‘unicode’ is a slightly slower method that works on any characters. None (default) does nothing.
**encoding** *(string, optional) –* If bytes are given, this encoding is used to decode. Default is ‘utf-8’.

**decode_error** *(('strict', 'ignore', 'replace'), optional) – Instruction on what to do if a byte Series is given that contains characters not of the given encoding. By default, it is ‘strict’, meaning that a UnicodeDecodeError will be raised. Other values are ‘ignore’ and ‘replace’.*

---

**Example**

```python
>>> import pandas
>>> from recordlinkage.standardise import clean

>>> name = ['Mary-ann', 'Bob :)', 'Angel', 'Bob (alias Billy)', None]
>>> s = pandas.Series(names)
>>> print(clean(s))
0  mary ann
1       bob
2       angel
3        bob
4         NaN
```

**Returns** `pandas.Series` – A cleaned Series of strings.

### phonenumbers

Clean phonenumbers by removing all non-numbers (except +).

**Parameters**
- **s** *(pandas.Series)* – A Series to clean.

**Returns** `pandas.Series` – A Series with cleaned phonenumbers.

### value_occurence

Count the number of times each value occurs.

This function returns the counts for each row, in contrast with `pandas.value_counts`.

**Returns** `pandas.Series` – A Series with value counts.

### phonetic

Convert names or strings into phonetic codes.

The implemented algorithms are `soundex`, `nysiis`, `metaphone` or `match_rating`.

**Parameters**
- **method** *(string)* – The algorithm that is used to phonetically encode the values. The possible options are “soundex”, “nysiis”, “metaphone” or “match rating”.
- **concat** *(bool, optional)* – Remove whitespace before phonetic encoding.
- **encoding** *(string, optional)* – If bytes are given, this encoding is used to decode. Default is ‘utf-8’.
- **decode_error** *(('strict', 'ignore', 'replace'), optional) – Instruction on what to do if a byte Series is given that contains characters not of the given encoding. By default, it is ‘strict’, meaning that a UnicodeDecodeError will be raised. Other values are ‘ignore’ and ‘replace’.)*

**Returns** `pandas.Series` – A Series with phonetic encoded values.
Note: The ‘soundex’ and ‘nysiis’ algorithms use the package ‘jellyfish’. It can be installed with pip (pip install jellyfish).

Indexing

Migrating from recordlinkage<=0.8.2 to recordlinkage>=0.9? Click here

The indexing module is used to make pairs of records. These pairs are called candidate links or candidate matches. There are several indexing algorithms available such as blocking and sorted neighborhood indexing. See the following references for background information about indexation.

class FullIndex

Class to generate a ‘full’ index.

A full index is an index with all possible combinations of record pairs. In case of linking, this indexation method generates the cartesian product of both DataFrame’s. In case of deduplicating DataFrame A, this indexation method are the pairs defined by the upper triangular matrix of the A x A.

Note: This indexation method can be slow for large DataFrame’s. The number of comparisons scales quadratic. Also, not all classifiers work well with large numbers of record pairs were most of the pairs are distinct.

index(x, x_link=None)

Make an index of record pairs.

Use a custom function to make record pairs of one or two dataframes. Each function should return a pandas.MultiIndex with record pairs.

Parameters

- x (pandas.DataFrame) – A pandas DataFrame. When x_link is None, the algorithm makes record pairs within the DataFrame. When x_link is not empty, the algorithm makes pairs between x and x_link.

- x_link (pandas.DataFrame, optional) – A second DataFrame to link with the DataFrame x.

Returns pandas.MultiIndex – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.

class BlockIndex(on=None, left_on=None, right_on=None)

Make candidate record pairs that agree on one or more variables.

Returns all record pairs that agree on the given variable(s). This method is known as blocking. Blocking is an effective way to make a subset of the record space (A * B).

Parameters

- on (label, optional) – A column name or a list of column names. These columns are used to block on.

- left_on (label, optional) – A column name or a list of column names of dataframe A. These columns are used to block on. This argument is ignored when argument ‘on’ is given.
• **right_on** *(label, optional)* – A column name or a list of column names of dataframe B. These columns are used to block on. This argument is ignored when argument ‘on’ is given.

### Examples

In the following example, the record pairs are made for two historical datasets with census data. The datasets are named `census_data_1980` and `census_data_1990`.

```python
>>> indexer = recordlinkage.BlockIndex(on='first_name')
>>> indexer.index(census_data_1980, census_data_1990)

Index (x, x_link=None)
Make an index of record pairs.

Use a custom function to make record pairs of one or two dataframes. Each function should return a pandas.MultiIndex with record pairs.

**Parameters**

- **x** *(pandas.DataFrame)* – A pandas DataFrame. When `x_link` is None, the algorithm makes record pairs within the DataFrame. When `x_link` is not empty, the algorithm makes pairs between `x` and `x_link`.

- **x_link** *(pandas.DataFrame, optional)* – A second DataFrame to link with the DataFrame `x`.

**Returns** pandas.MultiIndex – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.

### SortedNeighbourhoodIndex

```python
class SortedNeighbourhoodIndex(on=None, left_on=None, right_on=None, window=3, sorting_key_values=None, block_on=[], block_left_on=[], block_right_on=[])

Make candidate record pairs with the SortedNeighbourhood algorithm.

This algorithm returns record pairs that agree on the sorting key, but also records pairs in their neighbourhood. A large window size results in more record pairs. A window size of 1 returns the blocking index.

The Sorted Neighbourhood Index method is a great method when there is relatively large amount of spelling mistakes. Blocking will fail in that situation because it excludes to many records on minor spelling mistakes.

**Parameters**

- **on** *(label)* – Specify the on to make a sorted index

- **window** *(int, optional)* – The width of the window, default is 3

- **sorting_key_values** *(array, optional)* – A list of sorting key values (optional).

- **block_on** *(label)* – Additional columns to use standard blocking on

- **block_left_on** *(label)* – Additional columns of the left dataframe to use standard blocking on.

- **block_right_on** *(label)* – Additional columns of the right dataframe to use standard blocking on
Examples

In the following example, the record pairs are made for two historical datasets with census data. The datasets are named `census_data_1980` and `census_data_1990`.

```python
>>> indexer = recordlinkage.SortedNeighbourhoodIndex(on='first_name', w=9)
>>> indexer.index(census_data_1980, census_data_1990)
```

```
index(x, x_link=None)
Make an index of record pairs.

Use a custom function to make record pairs of one or two dataframes. Each function should return a pandas.MultiIndex with record pairs.

Parameters
- **x** *(pandas.DataFrame)* – A pandas DataFrame. When `x_link` is None, the algorithm makes record pairs within the DataFrame. When `x_link` is not empty, the algorithm makes pairs between `x` and `x_link`.
- **x_link** *(pandas.DataFrame, optional)* – A second DataFrame to link with the DataFrame `x`.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.
```

```python
class RandomIndex(n, replace=True, random_state=None)
Class to generate random pairs of records.

This class returns random pairs of records with or without replacement. Use the `random_state` parameter to seed the algorithm and reproduce results. This way to make record pairs is useful for the training of unsupervised learning models for record linkage.

Parameters
- **n** *(int)* – The number of record pairs to return. In case `replace=False`, the integer `n` should be bounded by `0 < n <= n_max` where `n_max` is the maximum number of pairs possible.
- **replace** *(bool, optional)* – Whether the sample of record pairs is with or without replacement. Default: True
- **random_state** *(int or numpy.random.RandomState, optional)* – Seed for the random number generator (if int), or numpy RandomState object.

index(x, x_link=None)
Make an index of record pairs.

Use a custom function to make record pairs of one or two dataframes. Each function should return a pandas.MultiIndex with record pairs.

Parameters
- **x** *(pandas.DataFrame)* – A pandas DataFrame. When `x_link` is None, the algorithm makes record pairs within the DataFrame. When `x_link` is not empty, the algorithm makes pairs between `x` and `x_link`.
- **x_link** *(pandas.DataFrame, optional)* – A second DataFrame to link with the DataFrame `x`.

Returns *pandas.MultiIndex* – A pandas.MultiIndex with record pairs. Each record pair contains the index labels of two records.
Migrating

Version 0.9 of the Python Record Linkage Toolkit uses a new indexing API. The new indexing API uses a different syntax. With the new API, each algorithm has its own class. See the following example to migrate a blocking index:

Old (linking):

```python
cl = recordlinkage.Pairs(df_a, df_b)
cl.block('given_name')
```

New (linking):

```python
cl = recordlinkage.BlockIndex('given_name')
cl.index(df_a, df_b)
```

Old (deduplication):

```python
cl = recordlinkage.Pairs(df_a)
cl.block('given_name')
```

New (deduplication):

```python
cl = recordlinkage.BlockIndex('given_name')
cl.index(df_a)
```

Comparing

The `recordlinkage.Compare` class and its methods can be used to compare records pairs. Several comparison methods are included such as string similarity measures, numerical measures and distance measures.

```python
class Compare(pairs, df_a=None, df_b=None, low_memory=False, block_size=1000000, njobs=1, **kwargs)
```

Compare record pairs with the tools in this class.

Class to compare the attributes of candidate record pairs. The `Compare` class has several methods to compare data such as string similarity measures, numeric metrics and exact comparison methods.

**Parameters**

- `pairs` (`pandas.MultiIndex`) – A MultiIndex of candidate record pairs.
- `df_a` (`pandas.DataFrame`) – The first dataframe.
- `df_b` (`pandas.DataFrame`) – The second dataframe.
- `low_memory` (`bool`) – Reduce the amount of memory used by the `Compare` class. Default False.
- `block_size` (`int`) – The maximum size of data blocks. Default 1,000,000.

- `pairs` `pandas.MultiIndex` – The candidate record pairs.
- `df_a` `pandas.DataFrame` – The first DataFrame.
- `df_b` `pandas.DataFrame` – The second DataFrame.
vectors

*pandas.DataFrame* – The DataFrame with comparison data.

**Examples**

In the following example, the record pairs of two historical datasets with census data are compared. The datasets are named *census_data_1980* and *census_data_1990*. The *candidate_pairs* are the record pairs to compare. The record pairs are compared on the first name, last name, sex, date of birth, address, place, and income.

```python
>>> comp = recordlinkage.Compare(
    candidate_pairs, census_data_1980, census_data_1990
)
>>> comp.string('first_name', 'name', method='jarowinkler')
>>> comp.string('lastname', 'lastname', method='jarowinkler')
>>> comp.exact('dateofbirth', 'dob')
>>> comp.exact('sex', 'sex')
>>> comp.string('address', 'address', method='levenshtein')
>>> comp.exact('place', 'place')
>>> comp.numeric('income', 'income')
>>> print(comp.vectors.head())
```

The attribute *vectors* is the DataFrame with the comparison data. It can be called whenever you want.

**exact** *(s1, s2, agree_value=1, disagree_value=0, missing_value=0, name=None, store=True)*

Compare the record pairs exactly.

**Parameters**

- **s1** *(label, pandas.Series)* – Series or DataFrame to compare all fields.
- **s2** *(label, pandas.Series)* – Series or DataFrame to compare all fields.
- **agree_value** *(float, str, numpy.dtype)* – The value when two records are identical. Default 1. If ‘values’ is passed, then the value of the record pair is passed.
- **disagree_value** *(float, str, numpy.dtype)* – The value when two records are not identical.
- **missing_value** *(float, str, numpy.dtype)* – The value for a comparison with a missing value. Default 0.
- **name** *(label)* – The name of the feature and the name of the column.
- **store** *(bool)* – Store the result in the dataframe. Default True

**Returns** *pandas.Series* – A pandas series with the result of comparing each record pair.

**string** *(s1, s2, method='levenshtein', threshold=None, missing_value=0, name=None, store=True)*

Compare strings.

**Parameters**

- **s1** *(label, pandas.Series)* – Series or DataFrame to compare all fields.
- **s2** *(label, pandas.Series)* – Series or DataFrame to compare all fields.
- **threshold** *(float, tuple of floats)* – A threshold value. All approximate string comparisons higher or equal than this threshold are 1. Otherwise 0.
numeric(s1, s2, method='linear', offset, scale, origin=0, missing_value=0, name=None, store=True)

Compute the similarity of numeric values.

This method returns the similarity of two numeric values. The implemented algorithms are: ‘step’, ‘linear’, ‘exp’, ‘gauss’ or ‘squared’. In case of agreement, the similarity is 1 and in case of complete disagreement it is 0. The implementation is similar with numeric comparing in ElasticSearch, a full-text search tool. The parameters are explained in the image below (source ElasticSearch, The Definitive Guide)

Parameters

• **s1** (label, pandas.Series) – Series or DataFrame to compare all fields.
• **s2** (label, pandas.Series) – Series or DataFrame to compare all fields.
• **offset** (float) – The offset. See image above.
• **scale** (float) – The scale of the numeric comparison method. See the image above. This argument is not available for the ‘step’ algorithm.
• **origin** (str) – The shift of bias between the values. See image above.
• **missing_value** (numpy.dtype) – The value if one or both records have a missing value on the compared field. Default 0.
• **name** (*label*) – The name of the feature and the name of the column.

• **store** (*bool*) – Store the result in the dataframe. Default True

**Returns**  
*pandas.Series* – A pandas series with the result of comparing each record pair.

**Note:** Numeric comparing can be an efficient way to compare date/time variables. This can be done by comparing the timestamps.

```
geo(lat1, lng1, lat2, lng2, method='linear', offset, scale, origin=0, missing_value=0, name=None, store=True)
```

Compute the similarity of two WGS84 coordinates.

Compare the geometric (haversine) distance between two WGS- coordinates. The similarity algorithms are 'step', 'linear', 'exp', 'gauss' or 'squared'. The similarity functions are the same as in *recordlinkage.comparing.Compare.numeric()*

**Parameters**

- **lat1** (*pandas.Series, numpy.array, label/string*) – Series with Lat-coordinates

- **lng1** (*pandas.Series, numpy.array, label/string*) – Series with Lng-coordinates

- **lat2** (*pandas.Series, numpy.array, label/string*) – Series with Lat-coordinates

- **lng2** (*pandas.Series, numpy.array, label/string*) – Series with Lng-coordinates


- **offset** (*float*) – The offset. See Compare.numeric.

- **scale** (*float*) – The scale of the numeric comparison method. See Compare.numeric. This argument is not available for the ‘step’ algorithm.

- **origin** (*float*) – The shift of bias between the values. See Compare.numeric.

- **missing_value** (*numpy.dtype*) – The value for a comparison with a missing value. Default 0.

- **name** (*label*) – The name of the feature and the name of the column.

- **store** (*bool*) – Store the result in the dataframe. Default True.

**Returns**  
*pandas.Series* – A pandas series with the result of comparing each record pair.

```
date(self, s1, s2, swap_month_day=0.5, swap_months='default', missing_value=0, name=None, store=True)
```

Compare two dates.

**Parameters**

- **s1** (*pandas.Series, numpy.array, label/string*) – Dates. This can be a Series, DatetimeIndex or DataFrame (with columns ‘year’, ‘month’ and ‘day’).

- **s2** (*pandas.Series, numpy.array, label/string*) – This can be a Series, DatetimeIndex or DataFrame (with columns ‘year’, ‘month’ and ‘day’).

- **swap_month_day** (*float*) – The value if the month and day are swapped.
• **swap_months** *(list of tuples)* – A list of tuples with common errors caused by the translating of months into numbers, i.e. October is month 10. The format of the tuples is `(month_good, month_bad, value)`. Default: `swap_months = [(6, 7, 0.5), (7, 6, 0.5), (9, 10, 0.5), (10, 9, 0.5)]`

• **missing_value** *(numpy.dtype)* – The value for a comparison with a missing value. Default 0.

• **name** *(label)* – The name of the feature and the name of the column.

• **store** *(bool)* – Store the result in the dataframe. Default True.

**Returns** *pandas.Series* – A pandas series with the result of comparing each record pair.

**clear_memory()**

Clear memory.

Clear some memory when `low_memory` was set to True.

**compare** *(comp_func, labels_a, labels_b, *args, **kwargs)*

Compare two records.

Core method to compare record pairs. This method takes a function and data from both records in the record pair. The data is compared with the compare function. The built-in methods also use this function.

**Example**

```python
>>> comp = recordlinkage.Compare(PAIRS, DATAFRAME1, DATAFRAME2)
>>> comp.exact('first_name', 'name')

>>> # same as
>>> comp.compare(recordlinkage._compare_exact, 'first_name', 'name')
```

**Parameters**

• **comp_func** *(function)* – A comparison function. This function can be a built-in function or a user defined comparison function.

• **labels_a** *(label, pandas.Series, pandas.DataFrame)* – The labels, Series or DataFrame to compare.

• **labels_b** *(label, pandas.Series, pandas.DataFrame)* – The labels, Series or DataFrame to compare.

• **name** *(label)* – The name of the feature and the name of the column.

• **store** *(bool, default True)* – Store the result in the dataframe.

**Returns** *pandas.Series* – A pandas series with the result of comparing each record pair.

**run()**

Run in a batch

This method is decrecated. Use the `comparing.Compare(..., low_memory=False)` for better performance.
Classification

Classification is the step in the record linkage process where record pairs are classified into matches, non-matches and possible matches [Christen2012]. Classification algorithms can be supervised or unsupervised (roughly speaking: with or without training data). Many of the algorithms need training data to classify the record pairs. Training data is data for which is known whether it is a match or not.

See also:

exception LearningError
  Learning error
class Classifier
  Base class for classification of records pairs.
  This class contains methods for training the classifier. Distinguish different types of training, such as supervised and unsupervised learning.
  learn (comparison_vectors, match_index, return_type='index')
    Train the classifier.
    Parameters
    • comparison_vectors (pandas.DataFrame) – The comparison vectors.
    • match_index (pandas.MultiIndex) – The true matches.
    • return_type ('index' (default), 'series', 'array') – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.
    Returns pandas.Series – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

predict (comparison_vectors, return_type='index')
  Predict the class of the record pairs.
  Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.
  Parameters
    • comparison_vectors (pandas.DataFrame) – Dataframe with comparison vectors.
    • return_type ('index' (default), 'series', 'array') – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.
    Returns pandas.Series – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

prob (comparison_vectors, return_type='series')
  Compute the probabilities for each record pair.
  For each pair of records, estimate the probability of being a match.
  Parameters
• **comparison_vectors** (*pandas.DataFrame*) – The dataframe with comparison vectors.

• **return_type** ("series" or "array") – Return a pandas series or numpy array. Default ‘series’.

**Returns** *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

class **KMeansClassifier**

```
match_cluster_center=None, nonmatch_cluster_center=None, *args, **kwargs)
```

KMeans classifier.

The K-means clustering algorithm (wikipedia) partitions candidate record pairs into matches and non-matches. Each comparison vector belongs to the cluster with the nearest mean. The K-means algorithm does not need trainings data, but it needs two starting points (one for the matches and one for the non-matches). The K-means clustering problem is NP-hard.

**Parameters**

• **match_cluster_center** (*list, numpy.array*) – The center of the match cluster. The length of the list/array must equal the number of comparison variables.

• **nonmatch_cluster_center** (*list, numpy.array*) – The center of the nonmatch (distinct) cluster. The length of the list/array must equal the number of comparison variables.

**classifier**

```
sklearn.cluster.KMeans – The Kmeans cluster class in sklearn.
```

**match_cluster_center**

```
list, numpy.array – The center of the match cluster.
```

**nonmatch_cluster_center**

```
list, numpy.array – The center of the nonmatch (distinct) cluster.
```

---

**Note:** There are way better methods for linking records than the k-means clustering algorithm. However, this algorithm does not need trainings data and is useful to do an initial partition.

**learn** (**comparison_vectors**, **return_type**=’index’)

Train the K-means classifier.

The K-means classifier is unsupervised and therefore does not need labels. The K-means classifier classifies the data into two sets of links and non-links. The starting point of the cluster centers are 0.05 for the non-matches and 0.95 for the matches.

**Parameters**

• **comparison_vectors** (*pandas.DataFrame*) – The dataframe with comparison vectors.

• **return_type** ("index" (default), "series", "array") – The format to return the classification result. The argument value ‘index’ will return the pandas.MultiIndex of the matches. The argument value ‘series’ will return a pandas.Series with zeros (distinct) and ones (matches). The argument value ‘array’ will return a numpy.ndarray with zeros and ones.

**Returns** *pandas.MultiIndex, pandas.Series or numpy.ndarray* – The prediction (see also the argument ‘return_type’).
predict (comparison_vectors, return_type='index')
Predict the class of the record pairs.
Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

• comparison_vectors (pandas.DataFrame) – Dataframe with comparison vectors.
• return_type ('index' (default), 'series', 'array') – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns pandas.Series – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

class LogisticRegressionClassifier (coefficients=None, intercept=None)
Logistic Regression Classifier.
This classifier is an application of the logistic regression model (wikipedia). The classifier partitions candidate record pairs into matches and non-matches.

Parameters

• coefficients (list, numpy.array) – The coefficients of the logistic regression.
• intercept (float) – The interception value.

classifier

coefficients
list – The coefficients of the logistic regression.

intercept
float – The interception value.

learn (comparison_vectors, match_index, return_type='index')
Train the classifier.

Parameters

• comparison_vectors (pandas.DataFrame) – The comparison vectors.
• match_index (pandas.MultiIndex) – The true matches.
• return_type ('index' (default), 'series', 'array') – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns pandas.Series – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

predict (comparison_vectors, return_type='index')
Predict the class of the record pairs.
Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.
Parameters

- **comparison_vectors** (*pandas.DataFrame*) – Dataframe with comparison vectors.

- **return_type** ('index' (default), 'series', 'array') – The format to return the classification result. The argument value ‘index’ will return the pandas.MultiIndex of the matches. The argument value ‘series’ will return a pandas.Series with zeros (distinct) and ones (matches). The argument value ‘array’ will return a numpy.ndarray with zeros and ones.

**Returns** *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

**prob** (*comparison_vectors*, **return_type**='series')

Compute the probabilities for each record pair.

For each pair of records, estimate the probability of being a match.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The dataframe with comparison vectors.

- **return_type** ('series' or 'array') – Return a pandas series or numpy array. Default ‘series’.

**Returns** *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

**class NaiveBayesClassifier** (*alpha=1.0*)

Naive Bayes Classifier.

The Naive Bayes classifier (wikipedia) partitions candidate record pairs into matches and non-matches. The classifier is based on probabilistic principles. The Naive Bayes classification method is proven to be mathematical equivalent with the Fellegi and Sunter model.

**Parameters**

- **log_prior** (*list, numpy.array*) – The log probability of each class.

- **classifier**


- **coefficients**

  *list* – The coefficients of the logistic regression.

- **intercept**

  *float* – The interception value.

**Parameters**

- **alpha** (*float*) – Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

**learn** (*comparison_vectors*, **match_index**, **return_type**='index')

Train the classifier.

Parameters

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors.

- **match_index** (*pandas.MultiIndex*) – The true matches.

- **return_type** ('index' (default), 'series', 'array') – The format to return the classification result. The argument value ‘index’ will return the pandas.MultiIndex of the matches. The argument value ‘series’ will return a pandas.Series
predict (comparison_vectors, return_type='index')

Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

**Parameters**

- **comparison_vectors** (*pandas.DataFrame*) – Dataframe with comparison vectors.
- **return_type** ('index' (default), 'series', 'array') – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

**Returns** *pandas.Series* – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

prob (comparison_vectors, return_type='series')

Compute the probabilities for each record pair.

For each pair of records, estimate the probability of being a match.

**Parameters**

- **comparison_vectors** (*pandas.DataFrame*) – The dataframe with comparison vectors.
- **return_type** ('series' or 'array') – Return a pandas series or numpy array. Default 'series'.

**Returns** *pandas.Series or numpy.ndarray* – The probability of being a match for each record pair.

class SVMClassifier

Support Vector Machines

The Support Vector Machine classifier (wikipedia) partitions candidate record pairs into matches and non-matches. This implementation is a non-probabilistic binary linear classifier. Support vector machines are supervised learning models. Therefore, the SVM classifiers needs training-data.

learn (comparison_vectors, match_index, return_type='index')

Train the classifier.

**Parameters**

- **comparison_vectors** (*pandas.DataFrame*) – The comparison vectors.
- **match_index** (*pandas.MultiIndex*) – The true matches.
- **return_type** ('index' (default), 'series', 'array') – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.
Returns `pandas.Series` – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

```python
predict(comparison_vectors, return_type='index')
```

Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- `comparison_vectors` (*pandas.DataFrame*) – Dataframe with comparison vectors.
- `return_type` (*'index' (default), 'series', 'array'*) – The format to return the classification result. The argument value ‘index’ will return the pandas.MultiIndex of the matches. The argument value ‘series’ will return a pandas.Series with zeros (distinct) and ones (matches). The argument value ‘array’ will return a numpy.ndarray with zeros and ones.

Returns `pandas.Series` – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

```python
class FellegiSunter(random_decision_rule=False)
```

Fellegi and Sunter framework.

Base class for probabilistic classification of records pairs with the Fellegi and Sunter (1969) framework.

```python
learn(comparison_vectors, match_index, return_type='index')
```

Train the classifier.

Parameters

- `match_index` (*pandas.MultiIndex*) – The true matches.
- `return_type` (*'index' (default), 'series', 'array'*) – The format to return the classification result. The argument value ‘index’ will return the pandas.MultiIndex of the matches. The argument value ‘series’ will return a pandas.Series with zeros (distinct) and ones (matches). The argument value ‘array’ will return a numpy.ndarray with zeros and ones.

Returns `pandas.Series` – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

```python
predict(comparison_vectors, return_type='index')
```

Predict the class of the record pairs.

Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

- `comparison_vectors` (*pandas.DataFrame*) – Dataframe with comparison vectors.
- `return_type` (*'index' (default), 'series', 'array'*) – The format to return the classification result. The argument value ‘index’ will return the pandas.MultiIndex of the matches. The argument value ‘series’ will return a pandas.Series with zeros (distinct) and ones (matches). The argument value ‘array’ will return a numpy.ndarray with zeros and ones.
Returns pandas.Series – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

prob(comparison_vectors, return_type='series')
Compute the probabilities for each record pair.
For each pair of records, estimate the probability of being a match.

Parameters

• comparison_vectors (pandas.DataFrame) – The dataframe with comparison vectors.
• return_type ('series' or 'array') – Return a pandas series or numpy array. Default 'series'.

Returns pandas.Series or numpy.ndarray – The probability of being a match for each record pair.

class ECMClassifier(*args, **kwargs)
Expectation/Conditional Maximisation classifier.

[EXPERIMENTAL] Expectation/Conditional Maximisation algorithm used as classifier. This probabilistic record linkage algorithm is used in combination with Fellegi and Sunter model.

learn(comparison_vectors, init='jaro', return_type='index')
Train the algorithm.
Train the Expectation-Maximisation classifier. This method is well-known as the ECM-algorithm implementation in the context of record linkage.

Parameters

• comparison_vectors (pandas.DataFrame) – The dataframe with comparison vectors.
• params_init (dict) – A dictionary with initial parameters of the ECM algorithm (optional).
• return_type ('index' (default), 'series', 'array') – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.

Returns pandas.Series – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

predict(comparison_vectors, return_type='index', *args, **kwargs)
Predict the class of record pairs.
Classify a set of record pairs based on their comparison vectors into matches, non-matches and possible matches. The classifier has to be trained to call this method.

Parameters

• comparison_vectors (pandas.DataFrame) – The dataframe with comparison vectors.
• return_type ('index' (default), 'series', 'array') – The format to return the classification result. The argument value 'index' will return the pandas.MultiIndex of the matches. The argument value 'series' will return a pandas.Series with zeros (distinct) and ones (matches). The argument value 'array' will return a numpy.ndarray with zeros and ones.
Returns pandas.Series – A pandas Series with the labels 1 (for the matches) and 0 (for the non-matches).

Note: Prediction is risky for this unsupervised learning method. Be aware that the sample from the population is valid.

prob(comparison_vectors)
Compute the probabilities for each record pair.
For each pair of records, estimate the probability of being a match.

Parameters
- comparison_vectors (pandas.DataFrame) – The dataframe with comparison vectors.
- return_type ('series' or 'array') – Return a pandas series or numpy array. Default ‘series’.

Returns pandas.Series or numpy.ndarray – The probability of being a match for each record pair.

Evaluation

Evaluation of classifications plays an important role in record linkage. Express your classification quality in terms accuracy, recall and F-score based on true positives, false positives, true negatives and false negatives.

reduction_ratio(n, x)
Compute the reduction ratio.
The reduction ratio is 1 minus the ratio candidate matches and the maximum number of pairs possible.

Parameters
- n (int, pandas.MultiIndex) – The number of candidate record pairs or the pandas.MultiIndex with record pairs.
- x (a list of pandas.DataFrame objects) – The data used to make the candidate record pairs.

Returns float – The reduction ratio.

max_pairs(shape)
Compute the maximum number of record pairs possible.

true_positives(true_match_index, matches_index)
Count the number of True Positives.
Return the number of correctly classified links, also called the number of True Positives (TP).

Parameters
- true_match_index (pandas.MultiIndex) – The golden/true links.
- match_index (pandas.MultiIndex) – The classified links.

Returns int – The number of correctly classified links.
**true_negatives** \((true\_match\_index, \text{matches\_index}, n\_pairs)\)

Count the number of True Negatives.

Return the number of correctly classified non-links, also called the number of True Negatives (TN).

**Parameters**

- **true_match_index** \((\text{pandas.MultiIndex})\) – The golden/true links.
- **match_index** \((\text{pandas.MultiIndex})\) – The classified links.
- **n_pairs** \((\text{int})\) – The number of candidate record pairs.

**Returns** \(\text{int}\) – The number of correctly classified non-links.

**false_positives** \((true\_match\_index, \text{matches\_index})\)

Count the number of false positives.

Return the number of predicted links, while the record pairs belong to different entities. This values is known as the number of False Positives (FP).

**Parameters**

- **true_match_index** \((\text{pandas.MultiIndex})\) – The golden/true links.
- **match_index** \((\text{pandas.MultiIndex})\) – The classified links.

**Returns** \(\text{int}\) – The number of false positives.

**false_negatives** \((true\_match\_index, \text{matches\_index})\)

Count the number of False Negatives.

Return the number of predicted non-links, while the record pairs belong to the same entity. This values is known as the number of False Negatives (FN).

**Parameters**

- **true_match_index** \((\text{pandas.MultiIndex})\) – The golden/true links.
- **match_index** \((\text{pandas.MultiIndex})\) – The classified links.

**Returns** \(\text{int}\) – The number of false negatives.

**confusion_matrix** \((true\_match\_index, \text{matches\_index}, n\_pairs)\)

Compute the confusion matrix.

The confusion matrix is of the following form:

<table>
<thead>
<tr>
<th></th>
<th>Predicted Positive</th>
<th>Predicted Negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Positive</td>
<td>True Positives (TP)</td>
<td>False Negatives (FN)</td>
</tr>
<tr>
<td>True Negative</td>
<td>False Positives (FP)</td>
<td>True Negatives (TN)</td>
</tr>
</tbody>
</table>

The confusion matrix is used to compute measures like precision and recall.

**Parameters**

- **true_match_index** \((\text{pandas.MultiIndex})\) – The golden/true links.
- **match_index** \((\text{pandas.MultiIndex})\) – The classified links.
- **n_pairs** \((\text{int})\) – The number of record pairs analysed.

**Returns** \(\text{numpy.array}\) – The confusion matrix with TP, TN, FN, FP values.

**precision** \((\text{confusion_matrix})\)

Compute the precision.

The precision is given by tp/(tp+fp).
Parameters `confusion_matrix` *(numpy.array)* – The matrix with tp, fn, fp, tn values.

Returns `float` – The precision

**recall** *(confusion_matrix)*
Compute the recall/sensitivity.
The recall is given by tp/(tp+fn).

Parameters `confusion_matrix` *(numpy.array)* – The matrix with tp, fn, fp, tn values.

Returns `float` – The recall

**accuracy** *(confusion_matrix)*
Compute the accuracy.
The accuracy is given by (tp+tn)/(tp+fp+tn+fn).

Parameters `confusion_matrix` *(numpy.array)* – The matrix with tp, fn, fp, tn values.

Returns `float` – The accuracy

**specificity** *(confusion_matrix)*
Compute the specificity.
The specificity is given by tn/(fp+tn).

Parameters `confusion_matrix` *(numpy.array)* – The matrix with tp, fn, fp, tn values.

Returns `float` – The specificity

**f1score** *(confusion_matrix)*
Compute the f1-score.
The f1-score is given by 2*(precision*recall)/(precision+recall).

Parameters `confusion_matrix` *(float)* – The matrix with tp, fn, fp, tn values.

Returns `float` – The f1-score

---

**Note:** If there are no pairs classified as links, this measure will raise a ZeroDivisionError.

---

**Datasets**

The Python Record Linkage Toolkit contains several open public datasets. Four datasets were generated by the developers of Febrl. In the future, we are developing tools to generate your own datasets.

**load_krebsregister** *(block=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], missing_values=None, shuffle=True)*

Dataset ‘Krebsregister’

This dataset of comparison patterns was obtained in an epidemiological cancer study in Germany. The comparison patterns were created by the Institute for Medical Biostatistics, Epidemiology and Informatics (IMBEI) and the University Medical Center of Johannes Gutenberg University (Mainz, Germany). The dataset is available for research online.

“The records represent individual data including first and family name, sex, date of birth and postal code, which were collected through iterative insertions in the course of several years. The comparison patterns in this data set are based on a sample of 100,000 records dating from 2005 to 2008. Data pairs were classified as “match” or “non-match” during an extensive manual review where several documentarists were involved. The resulting classification formed the basis for assessing the quality of the registry’s own record linkage procedure.”
In order to limit the amount of patterns a blocking procedure was applied, which selects only record pairs that meet specific agreement conditions. The results of the following six blocking iterations were merged together:

1. Phonetic equality of first name and family name, equality of date of birth.
2. Phonetic equality of first name, equality of day of birth.
3. Phonetic equality of first name, equality of month of birth.
4. Phonetic equality of first name, equality of year of birth.
5. Equality of complete date of birth.
6. Phonetic equality of family name, equality of sex.

This procedure resulted in 5,749,132 record pairs, of which 20,931 are matches. The data set is split into 10 blocks of (approximately) equal size and ratio of matches to non-matches.

**Parameters**

- **block (int, list)** – An integer or a list with integers between 1 and 10. The blocks are the blocks explained in the description.
- **missing_values (object, int, float)** – The value of the missing values. Default NaN.
- **shuffle (bool)** – Shuffle the record pairs. Default True.

**Returns** (pandas.DataFrame, pandas.MultiIndex) – A data frame with comparison vectors and a multi index with the indices of the matches.

**load_febrl1**

FEBRL dataset 1

The Freely Extensible Biomedical Record Linkage (Febrl) package was distributed with a dataset generator and four datasets generated with the generator. This functions returns the first Febrl dataset as a pandas DataFrame.

“This data set contains 1000 records (500 original and 500 duplicates, with exactly one duplicate per original record.”

**Returns** pandas.DataFrame – A pandas DataFrame with Febrl dataset 1.csv.

**load_febrl2**

FEBRL dataset 2

The Freely Extensible Biomedical Record Linkage (Febrl) package was distributed with a dataset generator and four datasets generated with the generator. This functions returns the second Febrl dataset as a pandas DataFrame.

“This data set contains 5000 records (4000 originals and 1000 duplicates), with a maximum of 5 duplicates based on one original record (and a poisson distribution of duplicate records). Distribution of duplicates: 19 original records have 5 duplicate records 47 original records have 4 duplicate records 107 original records have 3 duplicate records 141 original records have 2 duplicate records 114 original records have 1 duplicate record 572 original records have no duplicate record”

**Returns** pandas.DataFrame – A pandas DataFrame with Febrl dataset 2.csv.

**load_febrl3**

FEBRL dataset 3
The Freely Extensible Biomedical Record Linkage (Febrl) package was distributed with a dataset generator and four datasets generated with the generator. This function returns the third Febrl dataset as a pandas DataFrame.

“This data set contains 5000 records (2000 originals and 3000 duplicates), with a maximum of 5 duplicates based on one original record (and a Zipf distribution of duplicate records). Distribution of duplicates: 168 originals records have 5 duplicate records 161 originals records have 4 duplicate records 212 originals records have 3 duplicate records 256 originals records have 2 duplicate records 368 originals records have 1 duplicate record 1835 originals records have no duplicate record”

Returns pandas.DataFrame – A pandas DataFrame with Febrl dataset3.csv.

def load_febrl4() -> pandas.DataFrame
    FEBRL dataset 4
    The Freely Extensible Biomedical Record Linkage (Febrl) package was distributed with a dataset generator and four datasets generated with the generator. This function returns the fourth Febrl dataset as a pandas DataFrame.

    “Generated as one data set with 10000 records (5000 originals and 5000 duplicates, with one duplicate per original), the originals have been split from the duplicates, into dataset4a.csv (containing the 5000 original records) and dataset4b.csv (containing the 5000 duplicate records) These two data sets can be used for testing linkage procedures.”

    Returns (pandas.DataFrame, pandas.DataFrame) – A pandas DataFrame with Febrl dataset4a.csv and a pandas dataframe with Febrl dataset4b.csv.

def binary_vectors(n, n_match, m=[0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9, 0.9], u=[0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1], random_state=None) -> pandas.DataFrame
    Generate random binary comparison vectors

    This function is used to generate random comparison vectors. The result of each comparison is a binary value (0 or 1).

    Parameters
    - n (int) – The total number of comparison vectors.
    - n_match (int) – The number of matching record pairs.
    - m(list, default [0.9] * 8, optional) – A list of m probabilities of each partially identifying variable. The m probability is the probability that an identifier in matching record pairs agrees.
    - u(list, default [0.9] * 8, optional) – A list of u probabilities of each partially identifying variable. The u probability is the probability that an identifier in non-matching record pairs agrees.
    - random_state (int or numpy.random.RandomState, optional) – Seed for the random number generator with an integer or numpy RandomState object.

    Returns pandas.DataFrame – A dataframe with comparison vectors.
Thanks for your interest in contributing to the Python Record Linkage Toolkit. There is a lot of work to do. The workflow for contributing:

- clone https://github.com/J535D165/recordlinkage.git
- Make a branch with your modifications/contributions
- Write tests
- Run all tests
- Do a pull request

**Testing**

Install Nose:

```
pip install nose nose-parameterized
```

Run the following command to test the package

```
nosetests
```

**Performance**

Performance is very important in record linkage. The performance is monitored for all serious modifications of the core API. The performance monitoring is performed with Airspeed Velocity (asv).

Install Airspeed Velocity:

```
pip install asv
```
Run the following command from the root of the repository to test the performance of the current version of the package:

```
asv run
```

Run the following command to test all versions since tag v0.6.0

```
asv run --skip-existing-commits v0.6.0..master
```
Version 0.9.0

• A new index API. The new index API is no longer a single class (`recordlinkage.Pairs(...)` with all the functionality in it. The new API is based on Tensorflow and FEBRL. With the new structure, it easier to parallelise the record linkage process. In future releases, this will be implemented natively. See the reference page for more information and migrating.

• Significant speed improvement of the Sorted Neighbourhood Indexing algorithm. Thanks to @perryvais (PR #32).

• The function `binary_comparisons` is renamed. The new name of the function is `binary_vectors`. Documentation added to RTD.

• Add unit tests to test the generation of random comparison vectors.

Version 0.8.1

• Issues solved with rendering docs on ReadTheDocs. Still not clear what is going on with the `autodoc_mock_imports` in the sphinx conf.py file. Maybe a bug in sphinx.

• Move six to dependencies.

• The reference part of the docs is split into separate subsections. This makes the reference better readable.

• The landing page of the docs is slightly changed.

Version 0.8.0

• Add additional arguments to the function that downloads and loads the krebsregister data. The argument `missing_values` is used to fill missing values. Default: nothing is done. The argument `shuffle` is used to shuffle the records. Default is True.
• Remove the lastest traces of the old package name. The new package name is ‘Python Record Linkage Toolkit’
• Better error messages when there are only matches or non-matches are passed to train the classifier.
• Add AirSpeedVelocity tests to test the performance.
• Compare for deduplication fixed. It was broken.
• Parameterized tests for the Compare class and its algorithms. Making use of nose-parameterized module.
• Update documentation about contributing.
• Bugfix/improvement when blocking on multiple columns with missing values.
• Fix bug #29 (https://github.com/J535D165/recordlinkage/issues/29). Package not working with pandas 0.18 and 0.17. Dropped support pandas 0.17 and fixed support for 0.18. Also added multi-dendency tests for TravisCI.

Version 0.7.2
• Incorrect name of the Levenshtein method in the string comparison method fixed.

Version 0.7.1
• Fix the generation of docs on ReadTheDocs.
• Installation issue fixed. Packages not found.
• Import issues solved.
Version 0.7.0

- Rename the package into ‘Python Record Linkage Toolkit’
- Remove similar_values function
- Remove gender imputation tool
- Updated algorithms for comparing numeric variables. The new algorithms can compute the similarity with kernels like gaussian, linear, squared and exponential. Tests for these numeric comparison algorithms are included.
- Better NaN handling for compare functions.
- Algorithm added to compare dates.
- Add tests for date comparing.
- Divide the Compare class into two classes.
- Add documentation about performance tricks and concepts.
- Replace the comparison algorithms to a submodule.
- Include six in the package
- Drop requests module and use builtin Python functions.
- Add metaphone phonetic algorithm.
- Add match rating string comparing algorithm.
- Manual parameter handling for logistic regression. The attributes are coefficients and intercept.
- Drop class BernoulliNBClassifier.
- Various documentation updates.
- Many small other updates.

Version 0.6.0

- Reformatting the code such that it follows PEP8.
- Add Travis-CI and codecov support.
- Switch to distributing wheels.
- Fix bugs with depreciated pandas functions. __sub__ is no longer used for computing the difference of Index objects. It is now replaced by INDEX.difference(OTHER_INDEX).
- Exclude pairs with NaN’s on the index-key in Q-gram indexing.
- Add tests for krebsregister dataset.
- Fix Python3 bug on krebsregister dataset.
- Improve unicode handling in phonetic encoding functions.
- Strip accents with the clean function.
- Add documentation
- Bug for random indexing with incorrect arguments fixed and tests added.
- Improved deployment workflow
• And much more

Version 0.5.0

• Batch comparing added. Significant speed improvement.
• rldatasets are now included in the package itself.
• Added an experimental gender imputation tool.
• Blocking and SNI skip missing values
• No longer need for different index names
• FEBRL datasets included
• Unit tests for indexing and comparing improved
• Documentation updated

Version 0.4.0

• Fixes a serious bug with deduplication.
• Fixes undesired behaviour for sorted neighbourhood indexing with missing values.
• Add new datasets to the package like Febrl datasets
• Move Krebsregister dataset to this package.
• Improve and add some tests
• Various documentation updates

Version 0.3.0

• Total restructure of compare functions (The end of changing the API is close to now.)
• Compare method numerical is now named numeric and fuzzy is now named string.
• Add haversine formula to compare geographical records.
• Use numexpr for computing numeric comparisons.
• Add step, linear and squared comparing.
• Add eye index method.
• Improve, update and add new tests.
• Remove iterative indexing functions.
• New add chunks for indexing functions. These chunks are defined in the class Pairs. If chunks are defined, then the indexing functions returns a generator with an Index for each element.
• Update documentation.
• Various bug fixes.
Version 0.2.0

- Full Python3 support
- Update the parameters of the Logistic Regression Classifier manually. In literature, this is often denoted as the ‘deterministic record linkage’.
- Expectation/Conditional Maximization algorithm completely rewritten. The performance of the algorithm is much better now. The algorithm is still experimental.
- New string comparison metrics: Q-gram string comparing and Cosine string comparing.
- New indexing algorithm: Q-gram indexing.
- Several internal tests.
- Updated documentation.
- BernoulliNBClassifier is now named NaiveBayesClassifier. No changes to the algorithm.
- Arguments order in compare functions corrected.
- Function to clean phone numbers
- Return the result of the classifier as index, numpy array or pandas series.
- Many bug fixes

Version 0.1.0

- Official release


Index

A
accuracy() (in module recordlinkage.measures), 54

B
binary_vectors() (in module recordlinkage.datasets), 56
BlockIndex (class in recordlinkage.indexing), 37

C
Classifier (class in recordlinkage.classifiers), 45
classifier (KMeansClassifier attribute), 46
classifier (LogisticRegressionClassifier attribute), 47
classifier (NaiveBayesClassifier attribute), 48
clean() (in module recordlinkage.standardise.cleaning), 35
clear_memory() (Compare method), 44
coefficients (LogisticRegressionClassifier attribute), 47
coefficients (NaiveBayesClassifier attribute), 48
Compare (class in recordlinkage.comparing), 40
compare() (Compare method), 44
confusion_matrix() (in module recordlinkage.measures), 53

D
date() (Compare method), 43
df_a (Compare attribute), 40
df_b (Compare attribute), 40

E
ECMClassifier (class in recordlinkage.classifiers), 51
exact() (Compare method), 41

F
false_negatives() (in module recordlinkage.measures), 53
false_positives() (in module recordlinkage.measures), 53
FellegiSunter (class in recordlinkage.classifiers), 53
f_score() (in module recordlinkage.measures), 54
FullIndex (class in recordlinkage.indexing), 37

G
geo() (Compare method), 43

I
index() (BlockIndex method), 38
index() (FullIndex method), 37
index() (RandomIndex method), 39
index() (SortedNeighbourhoodIndex method), 39
intercept (LogisticRegressionClassifier attribute), 47
intercept (NaiveBayesClassifier attribute), 48

K
KMeansClassifier (class in recordlinkage.classifiers), 46
learn() (Classifier method), 45
learn() (ECMClassifier method), 51
learn() (FellegiSunter method), 50
learn() (KMeansClassifier method), 46
learn() (LogisticRegressionClassifier method), 47
learn() (NaiveBayesClassifier method), 48
learn() (SVMClassifier method), 49
LearningError, 45
load_febrl1() (in module recordlinkage.datasets), 55
load_febrl2() (in module recordlinkage.datasets), 55
load_febrl3() (in module recordlinkage.datasets), 55
load_febrl4() (in module recordlinkage.datasets), 56
load_krebsregister() (in module recordlinkage.datasets), 54
LogisticRegressionClassifier (class in recordlinkage.classifiers), 47

M
match_cluster_center (KMeansClassifier attribute), 46
max_pairs() (in module recordlinkage.measures), 52

N
NaiveBayesClassifier (class in recordlinkage.classifiers), 48
nonmatch_cluster_center (KMeansClassifier attribute), 46
numeric() (Compare method), 42

P
pairs (Compare attribute), 40
phonenumbers() (in module recordlinkage.standardise.cleaning), 36
phonetic() (in module recordlinkage.standardise.encoding), 36
precision() (in module recordlinkage.measures), 53
predict() (Classifier method), 45
predict() (ECMClassifier method), 51
predict() (FellegiSunter method), 50
predict() (KMeansClassifier method), 46
predict() (LogisticRegressionClassifier method), 47
predict() (NaiveBayesClassifier method), 49
predict() (SVMClassifier method), 50
prob() (Classifier method), 45
prob() (ECMClassifier method), 52
prob() (FellegiSunter method), 51
prob() (LogisticRegressionClassifier method), 48
prob() (NaiveBayesClassifier method), 49

R
RandomIndex (class in recordlinkage.indexing), 39
recall() (in module recordlinkage.measures), 54
recordlinkage.classifiers (module), 45
recordlinkage.comparing (module), 40
recordlinkage.indexing (module), 37
recordlinkage.measures (module), 52
recordlinkage.standardise (module), 35
recordlinkage.standardise.cleaning (module), 35
recordlinkage.standardise.encoding (module), 36
reduction_ratio() (in module recordlinkage.measures), 52
run() (Compare method), 44

S
SortedNeighbourhoodIndex (class in recordlinkage.indexing), 38
specificity() (in module recordlinkage.measures), 54
string() (Compare method), 41
SVMClassifier (class in recordlinkage.classifiers), 49

T
ture_negatives() (in module recordlinkage.measures), 52
ture_positives() (in module recordlinkage.measures), 52

V
value_occurrence() (in module recordlinkage.standardise.cleaning), 36
vectors (Compare attribute), 40